

# Test-Driven Design: Ein einfaches Beispiel

Martin Wirsing

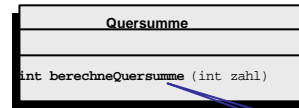
in Zusammenarbeit mit  
Matthias Hölzl, Piotr Kosiuczenko, Dirk Pattinson

05/03

Informatik II, SS 03

4

## UML Diagramm für Quersumme



Dieses UML-Diagramm spiegelt eine Entwurfsentscheidung wider; eine andere Möglichkeit wäre ein Attribut „zahl“ und int berechneQuersumme()

Berechnet Quersumme von zahl.

M. Wirsing: Spezifikation und Test

Informatik II, SS 03

2

## Ziele

- Veranschaulichung der Technik des Test-Driven Design am Beispiel eines Programms zur Berechnung der Quersumme
- Lernen mit Junit für Test-Driven Design einzusetzen

M. Wirsing: Spezifikation und Test

Informatik II, SS 03

5

## Das Gerüst der Testklasse Quersumme

```
import junit.framework.TestCase;

public class QuersummeTest extends TestCase
{
    public QuersummeTest()
    {
        super();
    }

    < Testmethoden >
    public static void main(String args[])
    {
        junit.textui.TestRunner.run(QuersummeTest.class);
    }
}
```

Kurze Iterationen, in denen abwechselnd Code und Test geschrieben wird.

M. Wirsing: Spezifikation und Test

Informatik II, SS 03

3

## Test-gesteuerter Entwurf

- Neue Software-Entwurfstechniken stellen das Testen vor das Implementieren des Programms:  
Externe Programming, Test-first Programming, Agile Software Development
- Schritte des Test-gesteuerten Entwurfs:
  1. Erstelle UML-Diagramm
  2. Entwerfe einen Test für eine Methode
  3. Schreibe möglichst einfachen Code, bis der Test nicht mehr fehlschlägt
  4. Wiederhole 2. und 3. bis alle Methoden des Klassendiagramms implementiert sind.
- Dabei wird häufig der Code (und manchmal der Test) restrukturiert („Refactoring“). Jedes Mal werden alle Tests durchgeführt, um sicher zu stellen, dass die ~~Code~~ Code restrukturierung nicht zu Fehlern im alten Code geführt hat.

Kurze Iterationen, in denen abwechselnd Code und Test geschrieben wird.

Tests für die „Grenzwerte“ 0 und 9.

M. Wirsing: Spezifikation und Test

Informatik II, SS 03

6

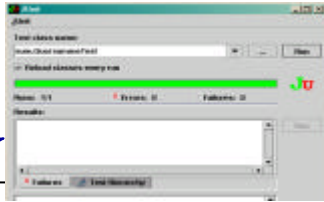
## Ein erster Test für Zahlen kleiner als 10

```
public void testQuersummeKleiner10()
{
    Quersumme o1 = new Quersumme();
    Quersumme o2 = new Quersumme();
    assertEquals(0, o1.berechneQuersumme(0));
    assertEquals(9, o2.berechneQuersumme(9));
}
```

M. Wirsing: Spezifikation und Test

### Ein möglichst einfaches Programm für den 1. Test

```
public class Quersumme
{
    public Quersumme(){}
    public int berechneQuersumme(int zahl)
    {
        return zahl;
    }
}
```



Gibt zahl als Wert zurück; dies ist ok für 0=zahl < 10!

Erwartungsgemäß erfüllt das einfache Programm diesen Test!

### Refactoring:

### Ein möglichst einfaches Programm für beide Tests

```
public class Quersumme
{
    public Quersumme(){}
    public int berechneQuersumme(int zahl)
    {
        if (zahl < 10) return zahl;
        else return berechneQuersumme(zahl/10)+zahl%10;
    }
}
```

Berechnet die Quersumme rekursiv. Was ist mit negativen Zahlen??

### Ein zweiter Test für Zahlen größer gleich 10

```
public void testQuersummeGroesserGleich10()
{
    Quersumme o1 = new Quersumme();
    Quersumme o2 = new Quersumme();
    assertEquals(1, o1.berechneQuersumme(10));
    assertEquals(11, o2.berechneQuersumme(19));
}
```

Grenzwert für die Fallunterscheidung GroesserGleich10

Ein beliebiger Wert größer als 10

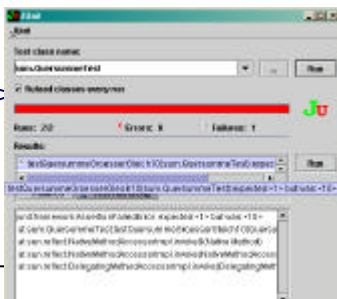
### Ein Test für negative Zahlen

```
public void testQuersummeKleiner0()
{
    Quersumme o1 = new Quersumme();
    Quersumme o2 = new Quersumme();
    assertEquals(-9, o1.berechneQuersumme(-9));
    assertEquals(-11, o2.berechneQuersumme(-19));
}
```

Grenzwert für die Fallunterscheidung GroesserGleich10

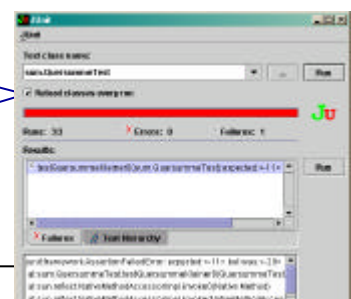
Ein beliebiger Wert größer als 10

### Der zweite Test für Zahlen größer gleich 10 schlägt fehl



Natürlich meldet JUnit Fehler. Das Programm muss geeignet restrukturiert werden.

### Der Test für negative Zahlen schlägt fehl!



Natürlich meldet JUnit Fehler. Das Programm muss geeignet restrukturiert werden.

### Refactoring: Ein erstes Programm für alle Tests

```
public class Quersumme
{
    public Quersumme(){}
    public int berechneQuersumme(int zahl)
    {
        if (zahl<10 & zahl>0) return zahl;
        else return berechneQuersumme(zahl/10)+zahl%10;
    }
}
```



Programmierfehler bei zahl = 0.

### Refactoring 2: Ein Schema zur Transformation linearer Rekursion in while-Programme

```
R f(T x)
{
    if (B(x)) return A(x);
    else return f(E(x)) C(x);
}

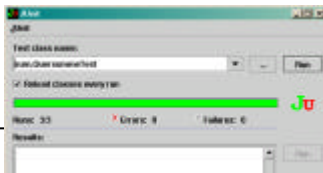
Ist äquivalent zu
R f(T x)
{
    R r = A(x);
    while (!(B(x)))
    {
        r = r C(x); x = E(x);
    }
    return r;
}
```

Falls kommutativ und assoziativ ist

Beispiel Quersumme:  
 A(x) entspricht +  
 B(x) entspricht x==0  
 C(x) entspricht x%10  
 E(x) entspricht x/10

### Verbesserung: Ein Programm für alle Tests

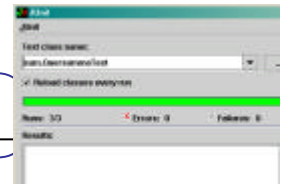
```
public class Quersumme
{
    public Quersumme(){}
    public int berechneQuersumme(int zahl)
    {
        if (zahl<10 & zahl>=0) return zahl;
        else return berechneQuersumme(zahl/10)+zahl%10;
    }
}
```



Jetzt laufen alle Tests!

### Refactoring 2: Ein iteratives Programm für alle Tests

```
public class Quersumme
{
    public Quersumme(){}
    public int berechneQuersumme(int zahl)
    {
        int qs = 0;
        while (zahl!=0)
        {
            qs = qs + zahl % 10;
            zahl = zahl / 10;
        }
        return qs;
    }
}
```



Berechnet die Quersumme iterativ. Erfüllt alle Tests!

### Refactoring 1: Algebraische Umformung

```
public int berechneQuersumme(int zahl)
{
    if (zahl<10 & zahl>=0) return zahl;
    else return berechneQuersumme(zahl/10)+zahl%10;
}

Da für 0<=zahl<10 gilt: zahl =
if (zahl==0) zahl else BerQS(zahl/10)+zahl%10
```

```
ist äquivalent zu
public int berechneQuersumme(int zahl)
{
    if (zahl==0) return zahl;
    else return berechneQuersumme(zahl/10)+zahl%10;
}
```

Auch hier laufen alle Tests!

### Zusammenfassung

- Beim Test-gesteuerten Entwurf werden zuerst die Tests entworfen und dann die Programme geschrieben.
- Programme werden immer wieder restrukturiert (Refactoring). Durch automatisches Testen ist dies gut machbar.
- Formale Umformungstechniken (algebraische Umformungen, Transformationstechniken) können sehr gut zum Refactoring eingesetzt werden.